Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System

Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, Martin Monperrus

Alleviating Patch Overfitting with Automatic Test Generation: A Study of Feasibility and Effectiveness for the Nopol Repair System

Zhongxing Yu · Matias Martinez · Benjamin Danglot · Thomas Durieux · Martin Monperrus

Abstract Among the many different kinds of program repair techniques, one widely studied family of techniques is called test suite based repair. However, test suites are in essence input-output specifications and are thus typically inadequate for completely specifying the expected behavior of the program under repair. Consequently, the patches generated by test suite based repair techniques can just overfit to the used test suite, and fail to generalize to other tests. We deeply analyze the overfitting problem in program repair and give a classification of this problem. This classification will help the community to better understand and design techniques to defeat the overfitting problem. We further propose and evaluate an approach called UnsatGuided, which aims to alleviate the overfitting problem for synthesis-based repair techniques with automatic test case generation. The approach uses additional automatically generated tests to strengthen the repair constraint used by synthesis-based repair techniques. We analyze the effectiveness of UnsatGuided: 1) analytically with respect to alleviating two different kinds of overfitting issues; 2) empirically based on an experiment over the 224 bugs of the Defects4J repository. The main result is that automatic test generation is effective in alleviating one kind of overfitting issue–regression introduction, but due to oracle problem, has minimal positive impact on alleviating the other kind of overfitting issue–incomplete fixing.

Keywords Program repair · Synthesis-based repair · Patch overfitting · Automatic test case generation

Zhongxing Yu . Benjamin Danglot . Thomas Durieux
Inria Lille - Nord Europe, Avenue du Halley, 59650 Villeneuve-d'Ascq, France
E-mail: zhongxing.yu@inria.fr . benjamin.danglot@inria.fr . thomas.durieux@inria.fr

Matias Martinez
University of Valenciennes, Malvache Building, Campus Mont Houy, 59313 Valenciennes Cedex 9, France
E-mail: matias.martinez@univ-valenciennes.fr

Martin Monperrus
School of Computer Science and Communication, KTH Royal Institute of Technology, Stockholm, Sweden
E-mail: martin.monperrus@csc.kth.se

1 Introduction

Automated program repair holds out the promise of saving debugging costs and patching buggy programs more quickly than humans. Given this great potential, there has been a surge of research on automated program repair in recent years and several different techniques have been proposed (Goues et al (2012); Nguyen et al (2013); Xuan et al (2016); Pei et al (2014); Long et al (2017)). These techniques differ in various ways, such as the kinds of used oracles and the fault classes they target[1] (Monperrus (2017)).

Among the many different techniques proposed, one widely studied family of techniques is called test suite based repair. Test suite based repair starts with some passing tests as the specification of the expected behavior of the program and at least one failing test as a specification of the bug to be repaired, and aims at generating patches that make all the tests pass. Depending the patch generation strategy, test suite based repair can further be informally divided into two general categories: generate-and-validate techniques and synthesis-based techniques. Generate-and-validate techniques use certain methods such as genetic programming to first generate a set of candidate patches, and then validate the generated patches against the test suite. Representative examples in this category include GenProg (Goues et al (2012)), PAR (Kim et al (2013)) and SPR (Long and Rinard (2015)). Synthesis-based techniques first use test execution information to build a repair constraint, and then use a constraint solver to synthesize a patch. Typical examples in this category include SemFix (Nguyen et al (2013)), Nopol (Xuan et al (2016)), and Angelix (Mechtaev et al (2016)). Empirical studies have shown the promise of test suite based repair techniques in tackling real-life bugs in real-life systems. For instance, GenProg (Goues et al (2012)) and Angelix (Mechtaev et al (2016)) can generate repairs for large-scale real-world C programs, while ASTOR (Martinez and Monperrus (2016)) and Nopol (Xuan et al (2016)) have given encouraging results (Martinez et al (2016)) on a set of real-life Java programs from the Defects4j benchmark (Just et al (2014a)).

However, test suites are in essence input-output specifications and are therefore typically inadequate for completely specifying the expected behavior. Consequently, the patches generated by test suite based program repair techniques pass the test suite, yet may be incorrect. The patches that are overly specific to the used test suite and fail to generalize to other tests are called overfitting patches (Smith et al (2015)). Overfitting indeed threats the validity of test suite based repair techniques and some recent studies have shown that a significant portion of the patches generated by test suite based repair techniques are overfitting patches (Smith et al (2015); Qi et al (2015); Martinez et al (2016); Le et al (2017b)).

In this paper, we deeply analyze the overfitting problem in program repair and identify two kinds of overfitting issues: incomplete fixing and regression introduction. Our empirical evaluation shows that both kinds of overfitting issues are common. Based on the overfitting issues that an overfitting patch has, we further define three kinds of overfitting patches. This characterization of overfitting will help the community to better understand the overfitting problem in program repair, and will hopefully guide the development of techniques for alleviating overfitting.

[1] In this paper, we use "fault" and "bug" interchangeably.

We further propose an approach called UnsatGuided, which aims to alleviate the overfitting problem for synthesis-based techniques. Given the recent significant progress in the area of automatic test generation, UnsatGuided makes use of automatic test case generation technique to obtain additional tests and then integrate the automatically generated tests into the synthesis process. The intuition behind UnsatGuided is that additional automatically generated tests can supplement the manually written tests to strengthen the repair constraint, and synthesis-based techniques can thus use the strengthened repair constraint to synthesize patches that suffer less from overfitting. To generate tests that can detect problems besides *crashes* and *uncaught exceptions*, state-of-art automatic test generation techniques generate tests that include assertions encoding the behavior observed during test execution on the current program. By using such automatic test generation techniques on the program to be repaired, some of the generated tests can possibly assert buggy behaviors and these tests with wrong oracles can mislead the synthesis process. UnsatGuided tries to identify and discard tests with likely wrong oracles through the idea that if the additional repair constraint from a generated test has a contradiction with the repair constraint established using the manually written test suite, then the generated test is likely to be a test with wrong oracle.

We analyze the effectiveness of UnsatGuided with respect to alleviating different kinds of overfitting issues. We then set up an empirical evaluation of UnsatGuided, which uses Nopol (Xuan et al (2016)) as the synthesis-based technique and EvoSuite (Fraser and Arcuri (2011)) as the automatic test case generation technique. The evaluation uses 224 bugs of the Defects4J repository (Just et al (2014a)) as benchmark. The results confirm our analysis and show that UnsatGuided 1) is effective in alleviating overfitting issue of regression introduction for 16/19 bugs; 2) does not break already correct patches; 3) can help a synthesis-based repair technique to generate additional correct patches.

To sum up, the contributions of this paper are:

- An analysis of the overfitting problem in automated program repair and a classification of overfitting.
- An approach, called UnsatGuided, to alleviate the overfitting problem for synthesis-based repair techniques.
- An analysis of the effectiveness of UnsatGuided in alleviating different kinds of overfitting issues, and the identification of deep limitations of using automatic test case generation to alleviate overfitting.
- An empirical evaluation of the prevalence of different kinds of overfitting issues on 224 bugs of the Defects4J repository, as well as an extensive evaluation of the effectiveness of UnsatGuided in alleviating the overfitting problem.

The remainder of this paper is structured as follows. We first present related work in Section 2. Section 3 first provides our analysis of the overfitting problem and the classification of overfitting issues and overfitting patches, then gives the algorithm of the proposed approach UnsatGuided, and finally analyzes the effectiveness of UnsatGuided. Section 4 presents an empirical evaluation of the prevalence of different kinds of overfitting issues and the effectiveness of UnsatGuided, followed by Section 5 which concludes this paper. This paper is a major revision of an Arxiv preprint (Yu et al (2017)).

2 Related Work

2.1 Program Repair

Due to the high cost of fixing bugs manually, there has been a surge of research on automated program repair in recent years. Automated program repair aims to correct software defects without the intervention of human developers, and many different kinds of techniques have been proposed recently. For a complete picture of the field, readers can refer to the survey paper (Monperrus (2017)). Generally speaking, automated program repair involves two steps. To begin with, it analyzes the buggy program and uses techniques such as genetic programming (Goues et al (2012)), program synthesis (Nguyen et al (2013)) and machine learning (Long and Rinard (2016)) to produce one or more candidate patches. Afterwards, it validates the produced candidate patches with an oracle that encodes the expected behavior of the buggy program. Typically used oracles include test suites (Goues et al (2012); Nguyen et al (2013)), pre- and post-conditions (Wei et al (2010)), and runtime assertions (Perkins et al (2009)). The proposed automatic program repair techniques can target different kinds of faults. While some automatic program techniques target the general types of faults and do not require the fault types to be known in advance, a number of other techniques can only be applied to specific types of faults, such as null pointer exception (Durieux et al (2017)), integer overflow (Brumley et al (2007)), buffer overflow (Shaw et al (2014)), memory leak (Gao et al (2015)), and error handling bugs (Tian and Ray (2017)).

2.2 Test Suite Based Program Repair

Among the various kinds of program repair techniques proposed, a most widely studied and arguably the standard family of techniques is called test suite based repair. The inputs to test suite based repair techniques are the buggy program and a test suite, which contains some passing tests as the specification of the expected behavior of the program and at least one failing test as a specification of the bug to be repaired. The output is one or more candidate patches that make all the test cases pass. Typically, test suite based repair techniques first use some fault localization techniques (Jones and Harrold (2005); Liu et al (2006); Yu et al (2015, 2013, 2011); Zhang et al (2006)) to identify the most suspicious program statements. Then, test suite based repair techniques use some patch generation strategies to patch the identified suspicious statements. Based on the used patch generation strategy, test suite based repair techniques can further be divided into generate-and-validate techniques and synthesis-based techniques.

Generate-and-validate repair techniques first search within a search space to generate a set of patches, and then validate them against the test suite. GenProg (Goues et al (2012)), one of the earliest generate-and-validate techniques, uses genetic programming to search the repair space and generates patches that consist of code snippets copied from elsewhere in the same program. PAR (Kim et al (2013)) shares the same search strategy with GenProg but uses 10 specialized patch templates derived from human-written patches to construct the search space. RSRepair (Qi et al (2014)) has the same search space as GenProg but uses random search instead, and the empirical evaluation shows that random search can

be as effective as genetic programming. AE (Weimer et al (2013)) employs a novel deterministic search strategy and uses program equivalence relation to reduce the patch search space. SPR (Long and Rinard (2015)) uses a set of predefined transformation schemas to construct the search space, and patches are generated by instantiating the schemas with condition synthesis techniques. Prophet (Long and Rinard (2016)) applies probabilistic models of correct code learned from successful human patches to prioritize candidate patches so that the correct patches could have higher rankings. Given that most of the proposed repair systems target only C code, jGenProg, as implemented in ASTOR (Martinez and Monperrus (2016)), is an implementation of GenProg for Java code.

Synthesis-based techniques first use the input test suite to extract a repair constraint, and then leverage program synthesis to solve the constraint and get a patch. The patches generated by synthesis-based techniques are generally by design correct with respect to the input test suite. SemFix (Nguyen et al (2013)), the pioneer work in this category of repair techniques, performs controlled symbolic execution on the input tests to get symbolic constraints, and uses code synthesis to identify a code change that makes all tests pass. The target repair locations of SemFix are assignments and boolean conditions. To make the generated patches more readable and comprehensible for human beings, DirectFix (Mechtaev et al (2015)) encodes the repair problem into a partial Maximum Satisfiability problem (MaxSAT) and uses a suitably modified Satisfiability Modulo Theory (SMT) solver to get the solution, which is finally converted into the concise patch. Angelix (Mechtaev et al (2016)) uses a lightweight repair constraint representation called "angelic forest" to increase the scalability of DirectFix. Nopol (Xuan et al (2016)) uses multiple instrumented test suite executions to synthesize a repair constraint, which is then transformed into a SMT problem and a feasible solution to the problem is finally returned as a patch. Nopol addresses the repair of buggy *if* conditions and missing preconditions. S3 (Le et al (2017a)) aims to synthesize more generalizable patches by using three components: a domain-specific language (DSL) to customize and constrain search space, an enumeration-based search strategy to search the space, and finally a ranking function to rank patches.

While test suite based repair techniques are promising, an inherent limitation of them is that the correctness specifications used by them are the test suites, which are generally available but rarely exhaustive in practice. As a result, the generated patches may just overfit to the available tests, meaning that they will break untested but desired functionality. Several recent studies have shown that overfitting is a serious issue associated with test suite based repair techniques. Qi et al. (Qi et al (2015)) find that the vast majority of patches produced by GenProg, RSRepair, and AE avoid bugs simply by functionality deletion. A subsequent study by Smith et al. (Smith et al (2015)) further confirms that the patches generated by GenProg and RSRepair fail to generalize. The empirical study conducted by Martinez et al. (Martinez et al (2016)) reveals that among the 47 bugs fixed by jGenProg, jKali, and Nopol, only 9 bugs are correctly fixed. More recently, the study by Le et al. (Le et al (2017b)) again confirms the severity of the overfitting issue for synthesis-based repair techniques. Moreover, the study also investigates how test suite size and provenance, number of failing tests, and semantics-specific tool settings can affect overfitting issues for synthesis-based repair techniques. Given the seriousness and importance of the overfitting problem, Yi et al. (Yi et al (2017)) explore the correlation between test suite metrics and the quality

of patches generated by automated program repair tetchiness, and they find that with the increase of traditional test suite metrics, the quality of the generated patches also tend to improve.

To gain a better understanding of the overfitting problem in program repair, we conduct a deep analysis of it and give the classification of overfitting issues and overfitting patches. We wish the classifications can facilitate future work on alleviating the overfitting problem in program repair. In addition, given the recent progress in the area of automatic test generation, we investigate the feasibility of augmenting the initial test suite with additional automatically generated tests to alleviate the overfitting problem. More specifically, we propose an approach called UnsatGuided, which aims to alleviate the overfitting problem for synthesis-based repair techniques. The effectiveness of UnsatGuided for alleviating different kinds of overfitting issues is analyzed and empirically verified, and we also point out the deep limitations of using automatic test generation to alleviate overfitting.

In the literature, there are several works that try to use test case generation to alleviate the overfitting problem in program repair. Xin and Reiss (Xin and Reiss (2017)) propose an approach to identify overfitting patches through test case generation, which generates new test inputs that focus on the semantic differences brought by the patches and relies on human beings to add oracles for the inputs. Yang et al. (Yang et al (2017)) aim to filter overfitting patches for generate-and-validate repair techniques through a framework named *Opad*, which uses fuzz testing to generate tests and relies on two inherent oracles, crash and memory-safety, to enhance validity checking of generated patches. By heuristically comparing the similarity of different execution traces, Liu et al. (Liu et al (2017)) also aim to identify overfitting patches generated by test suite based repair techniques. UnsatGuided is different from these works. On the one hand, these three works all try to use generated tests to identify overfitting patches generated by test suite based repair techniques and the generated tests are not used by the run of the repair algorithm itself. However, our aim is to improve the patch generated using manually written test suite and the generated tests are used by the repair algorithm to supplement the manually written test suite so that a better repair specification can be obtained. On the other hand, our work does not assume the specificity of the used oracle while the work by Xin and Reiss (Xin and Reiss (2017)) uses the human oracle and the work by Yang et al. (Yang et al (2017)) uses the crash and memory-safety oracles.

2.3 Automatic Test Case Generation

Despite tests are often created manually in practice, much research effort has been put on automated test generation techniques. In particular, a number of automatic test generation tools for mainstream programming languages have been developed over the past few years. These tools typically rely on techniques such as random test generation, search-based test generation and dynamic symbolic execution.

For Java, Randoop (Pacheco and Ernst (2007)) is the well-known random unit test generation tool. Randoop uses feedback-directed random testing to generate unit tests, and it works by iteratively extending method call sequences with randomly selected method calls and randomly selected arguments from previously constructed sequences. As Randoop test generation process uses a bottom-up ap-

proach, it cannot generate tests for a specific class. Other random unit test generation tools for Java include JCrasher (Csallner and Smaragdakis (2004)), CarFast (Park et al (2012)), T3 (Prasetya (2014)), TestFul (Baresi et al (2010)) and eToc (Tonella (2004)). There are also techniques that use various kinds of symbolic execution, such as symbolic PathFinder (Păsăreanu and Rungta (2010)) and DSC (Islam and Csallner (2010)). EvoSuite (Fraser and Arcuri (2011)) is the state-of-art search-based unit test generation tool for Java and can target a specific class. It uses an evolutionary approach to derive test suites that maximize code coverage, and generates assertions that encode the current behavior of the program.

In the C realm, DART (Godefroid et al (2005)), CUTE (Sen et al (2005)), and KLEE (Cadar et al (2008)) are three representatives of automatic test case generation tools for C. Symbolic execution is used in conjunction with concrete execution by these tools to maximize code coverage. In addition, Pex (Tillmann and De Halleux (2008)) is a popular unit test generation tool for C# code based on dynamic symbolic execution.

3 Analysis and Alleviation of the Overfitting Problem

In this section, we first introduce a novel classification of overfitting issues and overfitting patches. Then, we propose an approach called UnsatGuided for alleviating the overfitting problem for synthesis-based repair techniques. We finally analyze the effectiveness of UnsatGuided with respect to different overfitting kinds and point out the profound limitation of using automatic test generation to alleviate overfitting.

3.1 Core Definitions

Let us reason about the input space I of a program P. We consider modern object-oriented programs, where an input point is composed of one or more objects, interacting through a sequence of methods calls. In a typical repair scenario, the program is almost correct and thus a bug only affects the program behavior of a portion of the input domain, which we call the "buggy input domain" I_{bug}. We call the rest of the input domain, for which the program behaviors are considered correct as $I_{correct}$. By definition, a patch generated by an automatic program repair technique has an impact on program behaviors, i.e., it changes the behaviors of a portion of the input domain. We use I_{patch} to denote this input domain which is impacted by a patch. For input points within I_{bug} whose behaviors have been changed by a patch, the patch can either correctly or incorrectly change the original buggy behaviors. We use $I_{patch=}$ to denote the input points within I_{bug} whose behaviors have been incorrectly changed by a patch, i.e., the newly behaviors of these input points brought by the patch are still incorrect. Meanwhile, we use $I_{patch\checkmark}$ to denote the input points within I_{bug} whose behaviors have been correctly changed by a patch. If the patch involves changes to behaviors of input points within $I_{correct}$, then the original correct behaviors of these input points will undesirably become incorrect and we use $I_{patch\times}$ to denote these input points within $I_{correct}$ broken by the patch. Obviously, the union of $I_{patch=}$, $I_{patch\checkmark}$ and $I_{patch\times}$ makes up I_{patch}.

For simplicity, hereafter when we say some input points within I_{bug} are repaired by a patch, we mean the original buggy behaviors of these input points have been correctly changed by the patch. Similarly, when we say some input points within $I_{correct}$ are broken by a patch, we mean the original correct behaviors of these input points have been incorrectly changed by the patch.

Note as a patch generated by test suite based program repair techniques, the patch will at least repair the input points corresponding to the original failing tests. In other words, the intersection of $I_{patch✓}$ and I_{bug} will always not be empty ($I_{patch✓} \cap I_{correct} \neq \emptyset$).

3.2 Classification of Overfitting

For a given bug, a perfect patch repairs all input points within I_{bug} and does not break any input points within $I_{correct}$. However, due to the incompleteness of the test suite used to drive the repair process, the generated patch may not be ideal and just overfit to the used tests. Depending on how a generated patch performs with respect to the input domain I_{bug} and $I_{correct}$, we define two kinds of overfitting issues, which are consistent with the problems for human patches introduced by Gu et al (Gu et al (2010)).

Incomplete fixing: Some but not all input points within I_{bug} are repaired by the generated patch. In other words, $I_{patch✓}$ is a proper subset of I_{bug} ($I_{patch✓} \subset I_{bug}$).

Regression introduction: Some input points within $I_{correct}$ are broken by the generated patch. In other words, $I_{patch✗}$ is not an empty set ($I_{patch✗} \neq \emptyset$).

Based on these two different kinds of overfitting issues, we further define three different kinds of overfitting patches.

A-Overfitting patch: The overfitting patch only has the overfitting issue of incomplete fixing ($I_{patch✓} \subset I_{bug} \wedge I_{patch✗} = \emptyset$). This kind of overfitting patch can be considered as a "partial patch". It encompasses the worst case where there is one single failing test and the overfitting patch fixes the bug only for the input point specified in this specific failing test.

B-Overfitting patch: The overfitting patch only has the overfitting issue of regression introduction ($I_{patch✓} = I_{bug} \wedge I_{patch✗} \neq \emptyset$). Note that this kind of overfitting patch correctly repairs all input points within the buggy input domain I_{bug} but at the same time breaks some already correct behaviors of the buggy program under repair.

AB-Overfitting patch: The overfitting patch has both overfitting issues of incomplete fixing and regression introduction at the same time ($I_{patch✓} \subset I_{bug} \wedge I_{patch✗} \neq \emptyset$). This kind of overfitting patch correctly repairs some but not all input points within the buggy input domain I_{bug} and also introduces some regressions.

Figure 1 gives an illustration of these three different kinds of overfitting patches. This characterization of overfitting in program repair is independent of the technique presented in this paper and can be used by the community to better design techniques to defeat the overfitting problem.

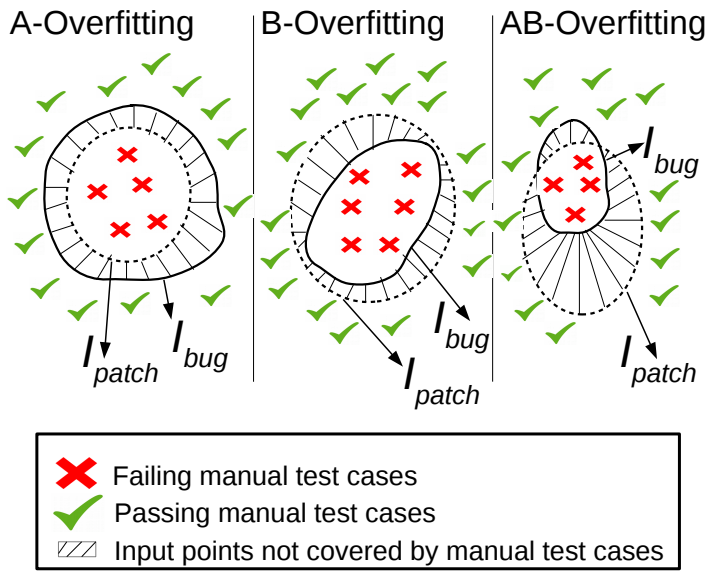

Fig. 1 A-Overfitting patch is a partial patch on a portion of the buggy input domain. B-Overfitting patch breaks correct behaviors outside the buggy input domain. AB-Overfitting patch partially fixes the buggy input domain and also breaks some correct behaviours.

3.3 UnsatGuided: Alleviating the Overfitting Problem for Synthesis-based Repair Techniques

In this section, we propose an approach called UnsatGuided, which aims to alleviate the overfitting problem for synthesis-based repair techniques. The approach aims to strengthen the correctness specification so that the resulting generated patches are more likely to generalize over the whole input domain. It achieves the aim by using additional tests generated by an automatic test case generation technique. We first give some background knowledge about automatic test case generation techniques and then give the details of the proposed approach.

3.3.1 The Bug-exposing Test Problem

In the context of regression testing, automatic test case generation techniques typically use the current behavior of the program itself as the oracle (Pacheco and Ernst (2007); Xie (2006))[2]. We consider those typical regression test generation techniques in this paper and denote an arbitrary technique as T_{reg}.

For a certain buggy version, T_{reg} may generate both input points within the buggy input domain I_{bug} and the correct input domain $I_{correct}$. For instance, suppose we have a calculator which incorrectly implements the `add` function for achieving the addition of two integers. The code is buggy on the input domain $(10, \_)$ (where _ means any integer except 0) and is implemented as follows:

[2] We do no uses the techniques that generate assertions from runs of different program versions (Taneja and Xie (2008); Evans and Savoia (2007)).

```
add(x,y) {
  if (x == 10) return x-y;
  else return x+y;
}
```

First, assume that T_{reg} generates a test in the correct input domain $I_{correct}$, say for input point $(5, 5)$. The resulting test, which uses the existing behavior as oracle, will be `assertEquals(10, add(5,5))`. Then consider what happens when the generated test lies in I_{bug}, say for input point $(10, 8)$. In this case, T_{reg} would generate the test `assertEquals(2, add(10,8))`.

If the input point of a generated test lies in I_{bug}, the synthesized assertion will assert the presence of the actual buggy behavior of the program under test, i.e., the generated assertion encodes the buggy behavior. In such a case, if the input point of a generated test lies in I_{bug}, it is called a "bug-exposing test" in this paper. Otherwise, the test is called a "normal test" if its input point lies in $I_{correct}$.

In the context of test suite based program repair, the existence of bug-exposing tests is a big problem. Basically, if a repair technique finds a patch that satisfies bug-exposing tests, then the buggy behavior is kept. In other words, it means that some of the generated tests can possibly enforce bad behaviors related with the bug to be repaired.

3.3.2 UnsatGuided: Incremental Test Suite Augmentation for Alleviating the Overfitting Problem for Synthesis-based Repair Techniques

The overfitting problem for synthesis-based repair techniques such as SemFix and Nopol arises because the repair constraint established using an incomplete test suite is not strong enough to fully express the intended semantics of a program. Our idea is to strengthen the initial repair constraint by augmenting the initial test suite with additional automatically generated tests. We wish that a stronger repair constraint would guide synthesis-based repair techniques towards better patches, i.e., patches that are correct or at least suffer less from overfitting.

The core problem to handle is the possible existence of bug-exposing test(s) among the tests generated by an automatic test case generation technique. We cannot directly supply all of the generated tests to a synthesis-based repair technique because bug-exposing tests can mislead the synthesis repair process and force incorrect behaviors to be synthesized.

To handle this core conceptual problem, we now present an approach called UnsatGuided, which gradually makes use of the new information provided by each automatically generated test to build a possibly stronger final repair constraint. The key underlying idea is that if the additional repair constraint enforced by an automatically generated test has logical contradictions with the repair constraint established so far, then the generated test is likely to be a bug-exposing test and is discarded.

Example To help understanding, we use the following toy program to illustrate it. The inputs are any integers and there is an error in the condition which results in buggy input domain $I_{bug} = \{5, 6, 7\}$. Suppose we use component based repair synthesis (Jha et al (2010)) to synthesize the correct condition, and to make the explanation easy, we further assume the available components include only variable x, the relational operators $<$ (less than) and $>$ (greater than), logical operator

&& (logical and), and finally any integer constants. For the three buggy inputs, regression test generation technique T_{reg} considered in this paper can generate bug-exposing tests `assertEquals(4, f(5))`, `assertEquals(5, f(6))`, and `assertEquals(6, f(7))`. Each test is of the form `assertEquals(`O`,`$f(I)$`)`, which specifies that the expected return value of the program is O when the input is I. For other input points, manually written tests and tests generated by T_{reg} are the same. Each test `assertEquals(`O`,`$f(I)$`)` will impose a repair constraint of the form $x{=}I{\rightarrow}f(I) = O$. The repair constraint imposed by a set of tests $\{t_i|\texttt{assertEquals(}O_i, f(I_i)\texttt{)}, 1 \leqslant i \leqslant N\}$ will be $\bigwedge_{i=1}^{N} (x{=}I_i{\rightarrow}f(I_i) = O_i)$. The repair constraint and available components are then typically encoded into a SMT problem, and a satisfying SMT model is then translated back into a synthesized expression which provably satisfies the repair constraint imposed by the tests. To achieve the encoding, techniques such as concrete execution (Xuan et al (2016)) and symbolic execution (Nguyen et al (2013)) can be used.

```
int f(int x) {
  if (x>0&&x<5) //faulty, correct condition should be (x>0&&x<8)
     x++;
  else
     x--;
  return x;
}
```

For this example, suppose the manually written tests `assertEquals(-1, f(0))`, `assertEquals(2, f(1))`, `assertEquals(8, f(7))`, and `assertEquals(9, f(10))` are provided initially. Using the repair constraint $(x = 0 \rightarrow f(0) = -1) \wedge (x = 1 \rightarrow f(1) = 2) \wedge (x = 7 \rightarrow f(7) = 8) \wedge (x = 10 \rightarrow f(10) = 9)$ enforced by these tests, the synthesis process can possibly synthesize a condition if (x>0 && x<10), which is not completely correct as the repair constraint enforced by the 4 manual tests is not strong enough. If a bug-exposing test such as `assertEquals(4,f(5))` is generated by T_{reg} and the repair constraint $(x = 5 \rightarrow f(5) = 4)$ imposed by it is added, the synthesis process cannot synthesize a condition as there is a contradiction between the repair constraint imposed by it and that imposed by the 4 manual tests. The contradiction happens because according to the the repair constraint imposed by manual tests and the available components used for synthesis, the calculation of any integer input between 1 and 7 should follow the same branch as integer inputs 1 and 7, consequently the return value should be 6 (not 4) when the integer input is 5. The core idea of UnsatGuided is to detect those contradictions and discard the bug exposing tests such as `assertEquals(4, f(5))`.

However, if a normal test such as `assertEquals(7, f(8))` is generated by T_{reg} and the repair constraint $(x = 8 \rightarrow f(8) = 7)$ imposed by it is added, there is no contradiction and a stronger repair constraint can be obtained, which will enable the synthesis process to synthesize the correct condition if (x>0 && x<8) in this specific example. The core idea of UnsatGuided is to keep those valuable new tests for synthesizing and validating patches.

Algorithm Algorithm 1 describes the approach in detail. The algorithm takes as input a buggy program P to be repaired, a manually written test suite TS which contains some passing tests and at least one failing test, a synthesis-based repair

Algorithm 1 : Algorithm for the Proposed Approach UnsatGuided

Input: A buggy program P and its manually written test suite TS
Input: A synthesis-based repair technique $T_{synthesis}$ and the time budget TB
Input: An automatic test case generation tool T_{auto}
Output: A patch pt to the buggy program P

```
1: pt_initial ← T_synthesis(P, TS, TB)
2: if pt_initial = null then
3:     pt ← null
4: else
5:     AGTS ← ∅
6:     pt ← pt_initial
7:     TS_aug ← TS
8:     t_initial ← getPatchGenTime(T_synthesis(P, TS, TB))
9:     {file_i}(i = 1, 2, ..., n) ← getInvolvedFiles(pt_initial)
10:    for i = 1 to n do
11:        AGTS ← AGTS ∪ T_auto(P, file_i)
12:    end for
13:    for j = 1 to |AGTS| do
14:        t_j ← AGTS(j)
15:        TS_aug ← TS_aug ∪ {t_j}
16:        pt_intern ← T_synthesis(P, TS_aug, t_initial × 2)
17:        if pt_intern ≠ null then
18:            pt ← pt_intern
19:        else
20:            TS_aug ← TS_aug − {t_j}
21:        end if
22:    end for
23: end if
24: return pt
```

technique $T_{synthesis}$, a time budget TB allocated for the execution of $T_{synthesis}$, and finally an automatic test case generation tool T_{auto} which uses a certain kind of automatic test case generation technique T_{reg}. The output of the algorithm is a patch pt to the buggy program P.

The algorithm directly returns an empty patch if $T_{synthesis}$ generates no patches within the time budget (lines 2-3). In case $T_{synthesis}$ generates an initial patch $pt_{initial}$ within the time budget, the algorithm first conducts a set of initialization steps as follows: it sets the automatically generated test suite $AGTS$ to be an empty set (line 5), sets the returned patch pt to be the initial patch $pt_{initial}$ (line 6), sets the augmented test suite TS_{aug} to be the manually written test suite TS (line 7), and gets the time used by $T_{synthesis}$ to generate the initial patch $pt_{initial}$ and sets $t_{initial}$ to be the value (line 8). Algorithm 1 then identifies the set of files $\{file_i\}$(i=1, 2,..., n) involved in the initial patch $pt_{initial}$ (line 9) and for each identified file, it uses the automatic test case generation tool T_{auto} to generate a set of tests that target behaviors related with the file and adds the generated tests to the automatically generated test suite $AGTS$ (lines 10-12).

Next, the algorithm will use the test suite $AGTS$ to refine the initial patch $pt_{initial}$. For each test t_j in the test suite $AGTS$ (line 14), the algorithm first adds it to the augmented test suite TS_{aug} (line 15) and runs technique $T_{synthesis}$ with test suite TS_{aug} and new time budget $t_{initial} \times 2$ against program P (line 16). The new time budget is used to quickly identify tests that can potentially contribute to strengthening the repair constraint, and thus improve the scalability of the approach. Then, if the generated patch pt_{intern} is not an empty patch,

the algorithm updates the returned patch pt with pt_{intern} (lines 17-18). In other words, the algorithm deems test t_j as a good test that can help improve the repair constraint. Otherwise, test t_j is removed from the augmented test suite TS_{aug} (lines 19-20) as t_j is either a bug-exposing test or it slows down the repair process too much. After the above process has been completed for each test in the test suite $AGTS$, the algorithm finally returns patch pt as the desirable patch (line 24).

Remark: Note for a certain synthesis-based repair technique $T_{synthesis}$ that is used as the input, UnsatGuided does not make any changes to the patch synthesis process of $T_{synthesis}$ itself. In particular, most current synthesis-based repair techniques use component based synthesis to synthesize the patch, including Nopol (Xuan et al (2016)), SemFix (Nguyen et al (2013)), Angelix (Mechtaev et al (2016)). For component-based synthesis, one important problem is selecting and using the build components. UnsatGuided keeps the original component selection and use strategy implemented by each synthesis-based repair technique.

In addition, the order of trying each test in the test suite $AGTS$ matters. Once a test is deemed as helpful, it is added to the augmented test suite TS_{aug} permanently and may impact the result of subsequent runs of other tests. The algorithm currently first uses the size of the identified files involved in the initial patch to determine the test generation order. The larger the size of an identified file, the earlier the test generation tool T_{auto} will generate tests for it. We first generate tests for big files as big files, in general, encode more logic compared to small files, thus tests generated for them are more important. Then, the algorithm uses the creation time of generated test files and the order of tests in a generated test file to prioritize tests. The earlier a test file is created, the earlier its test(s) will be tried by the algorithm. And if a test file contains multiple tests, the earlier a test appears in the file, the earlier the algorithm will try it. Future work will prioritize generated tests according to their potential to improve the repair constraint.

3.4 Analysis of UnsatGuided

UnsatGuided uses additional automatically generated tests to alleviate the overfitting problem for synthesis-based repair techniques. The performance of UnsatGuided is mainly affected by two aspects. On the one hand, it is affected by how the synthesis-based repair techniques perform with respect to the original manually written test suite, i.e., it depends on the overfitting type of the original patch. On the other hand, it is affected by whether or not the automatic test case generation technique generates bug-exposing tests. Let us dwell on this.

For ease of presentation, the initial repair constraint enforced by the manually written test suite is referred to as $RC_{initial}$, and the repair constraints enforced by the normal and bug-exposing tests generated by an automatic test case generation technique are referred to as RC_{normal} and RC_{buggy} respectively. Note due to the nature of test generation technique T_{reg}, RC_{buggy} is wrong. Also, we use $P_{original}$ to denote the original patch generated using the manually written test suite by a synthesis-based repair technique. Finally, we also use the example program in Section 3.3.2 to illustrate the key points of our analysis.

(1) $P_{original}$ **is correct**. In this case, $RC_{initial}$ is in general strong enough to drive the synthesis-based repair techniques to synthesize a correct patch. If the automatic test generation technique T_{reg} generates bug-exposing tests, RC_{buggy}

will have contradictions with $RC_{initial}$ (note RC_{buggy} is wrong) and UnsatGuided will recognize and discard these bug-exposing tests. Meanwhile, RC_{normal} is likely to be already covered by $RC_{initial}$ and is not likely to make $P_{original}$ become incorrect by definition. It can happen that the synthesis process coincidentally synthesizes a correct patch even though $RC_{initial}$ is weak, but this case is relatively rare. Thus, UnsatGuided generally will not change an already correct patch into an incorrect one.

For the example program in Section 3.3.2, suppose the manually written tests `assertEquals(-1, f(0))`, `assertEquals(2, f(1))`, `assert Equals(8, f(7))`, and `assertEquals(7, f(8))` are provided. In this case, the synthesis process can already use the repair constraint imposed by these 4 tests to synthesize the correct condition if (x>0 && x<8). Even if a bug-exposing test such as `assertEquals(4, f(5))` is generated, the repair constraint imposed by it will have a contradiction with the initial repair constraint (because it is impossible to synthesize a condition that satisfies the repair constraint imposed by all the 5 tests). Consequently, UnsatGuided will discard this bug-exposing test.

(2) $P_{original}$ **is A-overfitting**. In this case, $RC_{initial}$ is not strong enough to drive the synthesis-based repair techniques to synthesize a correct patch. More specifically, $RC_{initial}$ is in general strong enough to fully reflect the desired behaviors for correct input domain $I_{correct}$ but does not fully reflect the desired behaviors for all input points within buggy input domain I_{bug}. If the automatic test generation tool generates bug-exposing tests, the additional repair constraint enforced by a certain bug-exposing test does not necessarily have contradictions with $RC_{initial}$. If this happens, UnsatGuided is not able to identify and discard this kind of bug-exposing tests, and the synthesis process will be driven towards keeping the buggy behaviors corresponding to the bug-exposing tests. However, note this does not mean that the overfitting issue of incomplete fixing is worsened. If the behavior enforced by the kept bug-exposing test is already covered by the original patch, then it is likely that the synthesis process is not driven towards finding a new alternative solution and the overfitting issue of incomplete fixing remains the same. If the behavior enforced by the bug-exposing test is not covered by the original patch, then the synthesis process is likely to return a new solution. While the new solution indeed covers the new behavior enforced by the kept bug-exposing test, it can possibly generalize more over the whole I_{bug} compared to the original patch. Thus, the overfitting issue of incomplete fixing can both be worsened and improved if a new solution is returned. Meanwhile, the normal tests generated by T_{reg} by definition are not likely to be able to give additional repair constraints for input points within I_{bug}. Overall, for an A-overfitting patch, UnsatGuided is likely to have minimal positive impact and can coincidentally have a negative impact.

To illustrate, assume the provided manually written tests are `assertEquals (-1, f(0))`, `assertEquals(2,f(1))`, `assertEquals(7,f(6))`, and `assert Equals(7, f(8))` for the example program in Section 3.3.2. Using the repair constraint enforced by these tests, the synthesis process can possibly synthesize the condition if (x>0 && x<7), which is A-overfitting. Suppose bug-exposing test `assertE quals(4, f(5))` is generated, it will be discarded as the repair constraint imposed by it will make the synthesis process unable to synthesize a patch. However, if bug-exposing test `assertEquals(6, f(7))` is generated, it will be kept as there is no contradiction between the repair constraint enforced by it and

that enforced by the manual tests and the synthesis process can successfully return a patch. In this specific case, even though the bug-exposing test is kept, the synthesized patch is not likely to change as the behavior enforced by the bug-exposing test is already covered by the original patch. In other words, the overfitting issue of incomplete fixing remains the same as the original patch.

(3) $P_{original}$ **is B-overfitting**. In this case, $RC_{initial}$ is also not strong enough to drive the synthesis-based repair techniques to synthesize a correct patch. In particular, $RC_{initial}$ is in general strong enough to fully reflect the desired behaviors for buggy input domain I_{bug} but does not fully reflect the desired behaviors for all input points within correct input domain $I_{correct}$. In case the automatic test generation tool generates bug-exposing tests, RC_{buggy} is likely to have contradictions with $RC_{initial}$ (note $RC_{initial}$ is in general strong enough for input points within I_{bug}). Thus, UnsatGuided will identify and discard these bug-exposing tests. Meanwhile, RC_{normal} can supplement $RC_{initial}$ to better or even fully reflect the desired behaviors for input points within $I_{correct}$. Therefore, UnsatGuided can effectively help a B-overfitting patch reduce the overfitting issue of regression introduction, and can possibly turn a B-overfitting patch into a real correct one.

For the example program in Section 3.3.2, assume the manually written tests `assertEquals(-1, f(0))`, `assertEquals(2, f(1))`, `assertEquals(8, f(7))`, and `assertEquals(9, f(10))` are provided. Using the repair constraint enforced by these tests, the synthesis process can possibly synthesize the condition `if (x>0 && x<10)`, which is B-overfitting. If bug-exposing test `assertEquals(5, f(6))` is generated, UnsatGuided will discard it as the repair constraint imposed by it will make the synthesis process unable to synthesize a patch. If a normal test such as `assertEquals(8, f(9))` is generated by T_{reg}, it provides additional repair constraint for input points within $I_{correct}$ and can possibly help the synthesis process to synthesize the condition `if (x>0 && x<9)`, which has less overfitting issue of regression introduction compared to the original patch. In particular, if the normal test `assertEquals(7, f(8))` is generated by T_{reg}, this test will help the synthesis process to synthesize the exactly correct condition if (x>0 && x<8).

(4) $P_{original}$ **is AB-overfitting**. This case is a combination of case (2) and case (3). UnsatGuided can effectively help an AB-overfitting patch reduce the overfitting issue of regression introduction, but has minimal positive impact on reducing the overfitting issue of incomplete fixing. Note as bug-exposing tests by definition are not likely to give additional repair constraints for input points within the correct input domain $I_{correct}$, so the strengthened repair constraints for input points within $I_{correct}$ are not likely to be impacted even if some bug-exposing tests are generated and not removed by UnsatGuided. In other words, UnsatGuided will still be effective in alleviating overfitting issue of regression introduction.

Assume we have the manually written tests `assertEquals(-2, f(-1))`, `assertEquals(2, f(1))`, `assertEquals(7, f(6))`, and `assertEquals(7, f(8))` for the example program in Section 3.3.2. Using the repair constraint enforced by these tests, the synthesis process can possibly synthesis the condition if (x>-1 && x<7), which is AB-overfitting. If bug-exposing test `assertEquals(6, f(7))` and normal test `assertEquals(-1, f(0))` are generated, both of them will be kept and the synthesis process can possibly synthesize the condition if (x>0 && x<7), which has the same overfitting issue of incomplete fixing but less overfitting issue of regression introduction compared to the original patch.

In summary, UnsatGuided is not likely to break an already correct patch generated by a synthesis-based repair technique. For an overfitting patch, UnsatGuided can effectively reduce the overfitting issue of regression introduction, but has minimal positive impact on reducing the overfitting issue of incomplete fixing. With regard to turning an overfitting patch into a completely correct patch, UnsatGuided is likely to be effective only when the original patch generated using the manually written test suite is B-overfitting.

3.5 Discussion

We now discuss the general usefulness of automatic test generation in alleviating overfitting for synthesis-based repair techniques. The overall conclusion is for techniques that make use of automatically generated tests to strengthen the repair constraint, there exists a fundamental limitation which makes the above core limitation of just effectively reducing the overfitting issue of regression introduction general, i.e., not specific to the proposed technique UnsatGuided.

The fundamental limitation arises because of the oracle problem in automatic test generation. Due to the oracle problem, some of the automatically generated tests can encode wrong behaviors, which are called bug-exposing tests in this paper. Once the initial patch generated using the manually written test suite has the overfitting issue of incomplete fixing, the normal tests generated by an automatic test generation tool are not likely to be able to strengthen the repair constraints for input points within I_{bug}. While the bug-exposing tests generated by an automatic test generation tool can enforce additional repair constraints for input points within I_{bug}, the additional repair constraints enforced by bug-exposing tests are wrong. Different techniques can differ in how they classify automatically generated tests into normal tests and bug-exposing tests and how they further use these two kinds of tests, but they all face this fundamental problem. Consequently, for synthesis-based repair techniques, automatic test generation will not be very effective for alleviating the overfitting issue of incomplete fixing.

However, for the overfitting issue of regression introduction, the normal tests generated by an automatic test case generation tool can effectively supplement the manually written test suite to better build the repair constraints for input points within $I_{correct}$. By using the strengthened repair constraint, synthesis-based repair techniques can synthesize a patch that has less or even no overfitting issue of regression introduction. *According to this analysis, the usefulness of automatic test case generation in alleviating overfitting for synthesis-based repair techniques is mainly confined to reducing the overfitting issue of regression introduction.*

4 Experimental Evaluation

In this section, we present an empirical evaluation of the effectiveness of UnsatGuided in alleviating overfitting problems for synthesis-based repair techniques. In particular, we aim to empirically answer the following research questions:

– **RQ1**: How frequently do overfitting issues of incomplete fixing and regression introduction occur in practice for synthesis-based repair techniques?

Table 1 Descriptive Statistics of the 224 Considered Faults in Defects4J

Subjects	#Bugs	Source KLoC	Test KLoC	#Tests	Dev years
JFreechart	26	96	50	2,205	10
Commons Math	106	85	19	3,602	14
Joda-Time	27	28	53	4,130	14
Common Lang	65	22	6	2,245	15

- **RQ2**: How does UnsatGuided perform with respect to alleviating overfitting issues of incomplete fixing and regression introduction?
- **RQ3**: What is the impact of UnsatGuided on the correctness of the patches?
- **RQ4**: How does UnsatGuided respond to bug-exposing tests?
- **RQ5**: What is the time overhead of UnsatGuided?

4.1 Subjects of Investigation

4.1.1 Subject Programs

We selected Defects4J (Just et al (2014a)), a known database of real faults from real-world Java programs, as the experimental benchmark. Defects4J has different versions and the latest version of the benchmark contains 395 faults from 6 open source projects. Each fault in Defects4J is accompanied by a manually written test suite which contains at least one test that exposes the fault. In addition, Defects4J also provides commands to easily access faulty and fixed program versions for each fault, making it relatively easy to analyze them. Among the 6 projects, Mockito has been configured and added to the Defects4J framework recently (after we start the study presented in this paper). Thus we do not include the 38 faults for Mockito in our study. Besides, we also discard the 133 faults for Closure compiler as the tests are organized using scripts rather than the standard JUnit tests, which prevents these tests from running within our repair infrastructure. Consequently, we use the 224 faults of the remaining 4 projects in our experimental evaluation. Table 1 gives basic information about these 4 subjects.

4.1.2 Synthesis-based Repair Techniques

For our approach UnsatGuided to be implemented, we need a stable synthesis-based repair technique. In this study, Nopol (Xuan et al (2016)) is used as the representative of synthesis-based repair techniques. We select it for two reasons. First, Nopol is the only publicly-available synthesis-based repair technique that targets modern Java code. Second, it has been shown that Nopol is an effective automated repair system that can tackle real-life faults in real-world programs (Martinez et al (2016)).

4.1.3 Automatic Test Case Generation Tool

The automatic test case generation tool used in this study is EvoSuite (Fraser and Arcuri (2011)). EvoSuite aims to generate tests with maximal code coverage by

applying a genetic algorithm. Starting with a set of random tests, it then uses a coverage based fitness function to iteratively apply typical search operators such as selection, mutation, and crossover to evolve them. Upon finishing the search, it minimizes the test suite with highest code coverage with respect to the coverage criterion and adds regression test assertions. To our knowledge, EvoSuite is the state-of-art open source Java unit test generation tool. Compared with another popular test generation tool Randoop (Pacheco and Ernst (2007)), some recent studies (Almasi et al (2017); Shamshiri et al (2015)) have shown that Evosuite is better than Randoop in terms of a) compilable test generated, b) minimized flakiness, c) false positives, d) coverage, and e) most importantly–the number of bugs detected. While the generated tests by EvoSuite can possibly have problems of creating complex objects, exposing complex conditions, accessing private methods or fields, creating complex interactions, and generating appropriate assertions, they can be considered as effective in finding bugs in open-source and industrial systems in general (Shamshiri et al (2015)). Besides, as shown in algorithm 1, the approach UnsatGuided requires that the automatic test case generation tool is able to target a specific file of the program under repair. EvoSuite is indeed capable of generating tests for a specific class.

To generate more tests and make the test generation process itself as deterministic as possible, i.e., the generated tests should be the same if somebody else repeats out experiment, we made some changes about the timeout value, search budget value, sandboxing and mocking setting in the default EvoSuite option. The complete EvoSuite setting is available on Github.[3]

4.2 Experimental Setup

For each of the 224 studied faults in the Defects4J dataset, we run the proposed approach UnsatGuided against it. Whenever the test generation process is invoked, we run EvoSuite 30 times with different seeds to account for the randomness of EvoSuite following the guideline given in (Arcuri and Briand (2011)). The 30 seeds are 30 integer numbers randomly selected between 1 and 200. In addition, EvoSuite can generate tests that do not compile or generates tests that are unstable (i.e., tests which could fail or pass for the same configuration) due to the use of non-deterministic APIs such as date and time of day. Similar to the work in (Just et al (2014b); Shamshiri et al (2015)), we use the following process to remove the uncompilable and unstable tests if they exist:

(i) Remove all uncompilable tests;

(ii) Remove all tests that fail during re-execution on the program to be repaired;

(iii) Iteratively remove all unstable tests: we execute each compliable test suite on the program to be repaired five times consecutively. If any of these executions reveals unstable tests, we then remove these tests and re-compile and re-execute the test suite. This process is repeated until all remaining tests in the test suite pass five times consecutively.

Our experiment is extremely time-consuming. To make the time cost manageable, the timeout value for UnsatGuided, i.e., the input time budget in algorithm

[3] `https://github.com/Spirals-Team/test4repair-experiments`

1 for Nopol, is set to be 40 minutes in our experimental evaluation. Besides this change to global timeout value, we use the default configuration parameters of Nopol during its run. The experiment was run on a cluster consisting of 200 virtual nodes running Ubuntu 16.04 on a single Intel 2.68 GHz Xeon core with 1GB of RAM. As UnsatGuided will invoke the synthesis-based repair technique for each test generated, the whole repair process may still cost a lot of time. If so, we reduce the number of considered seeds. This happens for 2 faults (Chart_26 and Math_24), for which combining Nopol with UnsatGuided will generally cost more than 13 hours for each EvoSuite seed. Consequently, we use 10 seeds for these two bugs only for sake of time. Following an open-science ethics, all the code and data is made publicly available on the mentioned Github site in Section 4.1.3.

4.3 Evaluation Protocol

We evaluate the effectiveness of UnsatGuided from two points: its impact on the overfitting issue and correctness of the original patch generated by Nopol.

4.3.1 Assess Impact on Overfitting Issue

We have several major phases to evaluate the impact of UnsatGuided on overfitting issue of the original Nopol patch.

(1) *Test Case Selection and Classification.* To determine whether a patch has overfitting issue of incomplete fixing or regression introduction, we need to see whether the corresponding patched program will fail tests from buggy input domain I_{bug} or correct input domain $I_{correct}$ of the program to be repaired. As it is impractical to enumerate all tests from these two input domains, we view all tests generated for all seeds during our run of UnsatGuided (see Section 4.2) for a buggy program version as a representative subset of tests from these two input domains for this buggy program version in this paper. We believe it is reasonable from two aspects. On the one hand, we use a large number of seeds (30 in most cases) for each buggy program version, so we will have a large number of tests in general for each buggy program version. On the other hand, these tests all focus on testing the behaviors related with the patched highly suspicious files.

We then need to classify the generated tests as being in the buggy input domain or being in the correct input domain. Recall that during our run of UnsatGuided, EvoSuite uses the version-to-be-repaired as the oracle to generate tests. After the run of UnsatGuided for each seed, we thus have an EvoSuite test set which contain both 1) normal tests whose inputs are from $I_{correct}$ and the assertions of them are right, and 2) bug-exposing tests whose inputs are from I_{bug} and the assertions of them are wrong. To distinguish these two kinds of tests, we use the correct version of the version-to-be-repaired to achieve this goal. Note the assumption of the existence of a correct version is used here just for the evaluation purpose, we do not have this assumption for the run of UnsatGuided.

More specifically, given a buggy program P_{buggy}, the correct version $P_{correct}$ of P_{buggy}, and an EvoSuite test suite TS_{Evo_i} generated during the run of UnsatGuided for seed $seed_i$, we run TS_{Evo_i} against $P_{correct}$ to identify bug-exposing tests. As TS_{Evo_i} is generated from P_{buggy}, tests can possibly assert wrong behaviors. Thus, *a test fails over $P_{correct}$ is a bug-exposing test* and is added to

the test set $TS_{bugexpo}$. Otherwise, it is a normal test and is added to the test set TS_{normal}. For a certain buggy program version, this process is executed for each EvoSuite test suite TS_{Evo_j} generated for each seed $seed_j$ of the seed set $\{seed_j | 1 \leqslant j \leqslant N, N = 30 \text{ or } 10\}$. Consequently, for a specific buggy program version, $TS_{bugexpo}$ contains all bug-exposing tests and TS_{normal} contains all normal tests among all tests generated for all seeds during the run of UnsatGuided for this buggy program version.

(2) *Analyze the Overfitting Issue of the Synthesized Patches.* For a buggy program P_{buggy}, the correct version $P_{correct}$ of P_{buggy}, and the patch pc to P_{buggy}, we then use the identified test sets $TS_{bugexpo}$ and TS_{normal} in the previous step to analyze the overfitting issue of pc.

To determine whether patch pc has overfitting issue of regression introduction, we execute the program obtained by patching buggy program P_{buggy} with pc against TS_{normal}. If at least one test in TS_{normal} fails, then patch pc has overfitting issue of regression introduction.

To determine whether patch pc has overfitting issue of incomplete fixing, it is harder. The basic idea is executing the program obtained by patching buggy program P_{buggy} with pc against $TS_{bugexpo}$, and patch pc has overfitting issue of incomplete fixing if at least one test in $TS_{bugexpo}$ fails. However, recall that the tests in $TS_{bugexpo}$ are generated based on the buggy version P_{buggy}, i.e., the oracles are incorrect. Consequently, we first need to obtain the correct oracles for all tests in $TS_{bugexpo}$. We again use the correct version $P_{correct}$ to achieve this goal and the process is as follows.

First, for each failing assertion contained in a test from $TS_{bugexpo}$, we first capture the value it receives when the test is executed on the correct version $P_{correct}$. For instance, given a failing assertion `assertEquals(10,calculateValue(`y`))`, 10 is the value that the assertion expects and the value from `calculateValue(`y`)` is the received value. For this specific example, we need to capture the value for `calculateValue(`y`)` on $P_{correct}$ (note the value that P_{buggy} returns for `calculateValue(`y`)` is 10). Then, we replace the expected value in the failing assertion with the received value established on $P_{correct}$. For the previous example, if `calculateValue(`y`)` returns the value 5 on $P_{correct}$, the repaired assertion is `assertEquals(5, calculateValue(`y`))`.

The above process turns $TS_{bugexpo}$ into $TS_{bugexpo\checkmark}$ so that all bug-exposing tests will have correct oracles. After obtaining $TS_{bugexpo\checkmark}$, we run $TS_{bugexpo\checkmark}$ against the program obtained by patching buggy program P_{buggy} with pc. If we observe any failing tests, then patch pc has overfitting issue of incomplete fixing.

(3) *Measure Impact.* To evaluate the impact of UnsatGuided on the overfitting issue for a certain buggy program version, we compare the overfitting issue of the original Nopol patch $pc_{original}$ generated using the manually written test suite with that of the new patch pc_{new} generated after running UnsatGuided. More specifically, the process is as follows.

First, we use phases (1) and (2) to see whether the original patch $pc_{original}$ has overfitting issue of incomplete fixing or regression introduction. When we observe failing tests from TS_{normal} or $TS_{bugexpo\checkmark}$, we record the detailed number of failing tests. The recorded number represents the severity of the overfitting issue.

Second, for a patch pc_{new_i} generated by running UnsatGuided using a certain seed $seed_i$, we also use phases (1) and (2) to see whether the new patch pc_{new_i} has overfitting issue of incomplete fixing or regression introduction and record the

number of failing tests if we observe failing tests from TS_{normal} or $TS_{bugexpo✓}$. Note besides the test suite (corresponding to $seed_i$) used by UnsatGuided to generate pc_{new_i}, we also use all the other test suites generated for other seeds to evaluate the overfitting issue of pc_{new_i}.

Finally, the result obtained for pc_{new_i} is compared with that for $pc_{original}$ to determine the impact of UnsatGuided.

We repeat this process for each patch generated using each seed for a certain program version (i.e., the patch set $\{pc_{new\_i} \mid 1 \leqslant i \leqslant N, N = 30 \text{ or } 10\}$), and use the average result to assess the overall impact of UnsatGuided.

4.3.2 Assess Impact on Correctness

We compare the correctness of the patch generated after the run of UnsatGuided with that generated using Nopol to see the impact of UnsatGuided on patch correctness. To determine the correctness of a patch, the process is as follows.

First, we look at whether the generated tests reveal that there exist overfitting issues for a certain generated patch according to the procedure in Section 4.3.1.

Second, we manually analyze the generated patch and compare it with the corresponding human patch. A generated patch is deemed as correct only if it is exactly the same or semantically equivalent to the human patch. The equivalence is established based on the authors' understanding of the patch. To reduce the possible bias introduced as much as possible, two of the authors analyze the correctness of the patches separately and the results reported in this paper are based on the agreement between them. Note that the corresponding developer patches for several buggy versions trigger exceptions and emit text error messages if certain conditions are true, we count a generated patch correct if it triggers the same type of exceptions as the human patch under the exception conditions and we do not take the error message into account.

Note due to the use of different Nopol versions, the Nopol patches generated in this paper for some buggy versions are different from that generated in (Martinez et al (2016)). We thus replicate the manual analysis of the original Nopol patches.

As we use a large number of seeds (30 in most cases) for running UnsatGuided, it can happen that we have a large number of generated patches that are different from the original Nopol patch for a certain buggy version. For the inherent difficulty of the manual analysis, it is unrealistic to analyze all of the newly generated patches. To make the manual analysis realistic, for each buggy version, we randomly select one patch that is different from the original Nopol patch across all of the different kinds of patches generated for all seeds. It can happen that for a certain buggy version, the newly generated patches after the run of UnsatGuided for all seeds are the same as the original Nopol patch. In this case, it is obvious that UnsatGuided has no impact on the change of patch correctness.

4.4 Result Presentation

Table 2 displays the experimental results on combining Nopol with UnsatGuided (hereafter referred to as Nopol+UnsatGuided). This table only shows the Defects4J bugs that can be originally repaired by Nopol, and their identifiers are listed in column *Bug ID*.

Table 2 Experimental results with Nopol+UnsatGuided on the Defects4j Repository, only show bugs with test-suite adequate patches by plain Nopol.

Bug ID	Tests		Nopol			Nopol+UnsatGuided						
	#EvoTests	#Bug-expo	Time (hh:mm)	incomplete fix (#failing)	regression (#failing)	correctness	#Removed	#Removed Bug-expo	Avg #Time (hh:mm)	Change ratio (#unique)	fix completeness change (Avg #Removedinc)	regression change (Avg #Removedreg)	correctness
Chart_1	3012	0	00:02	No (0)	No (0)	NO	0	0	03:00	0/30 (1)	same (0)	same (0)	NO
Chart_5	2931	3	00:01	No (0)	Yes (10)	NO	104	3	01:18	27/30 (27)	same (0)	improve (2.9)	NO
Chart_9	3165	0	00:01	No (0)	No (0)	NO	0	0	01:00	0/30 (1)	same (0)	same (0)	NO
Chart_13	852	0	00:02	No (0)	No (0)	NO	0	0	00:24	30/30 (2)	same (0)	same (0)	NO
Chart_15	3711	0	00:04	No (0)	Yes (4)	NO	5	0	06:48	27/30 (23)	same (0)	improve (2.0)	NO
Chart_17	3246	10	00:01	Yes (10)	No (0)	NO	27	0	00:48	0/30 (1)	same (0)	same (0)	NO
Chart_21	1584	0	00:01	No (0)	Yes (6)	NO	0	0	00:48	30/30 (30)	same (0)	improve (6.0)⋆	NO
Chart_25	441	0	00:01	No (0)	Yes (8)	NO	0	0	00:12	8/30 (6)	same (0)	improve (8.0)⋆	NO
Chart_26	2432	0	00:03	No (0)	Yes (6)	NO	6	0	13:36	10/10 (5)	same (0)	improve (6.0)⋆	NO
Lang_44	3039	13	00:01	No (0)	No (0)	**YES**	13	13	00:48	3/30 (2)	same (0)	same (0)	**YES**
Lang_51	3720	1	00:01	No (0)	No (0)	NO	15	1	01:00	29/30 (2)	same (0)	same (0)	NO
Lang_53	2931	0	00:01	No (0)	No (0)	NO	0	0	00:06	26/30 (18)	same (0)	same (0)	NO
Lang_55	606	0	00:01	No (0)	No (0)	**YES**	1	0	00:12	30/30 (1)	same (0)	same (0)	**YES**
Lang_58	6471	0	00:01	No (0)	Yes (5)	NO	5	0	01:42	0/30 (1)	same (0)	same (0)	NO
Lang_63	1383	1	00:01	No (0)	No (0)	NO	33	1	00:36	27/30 (5)	same (0)	same (0)	NO
Math_7	876	2	00:16	Yes (2)	No (0)	NO	0	0	05:00	2/30 (3)	same (0)	same (0)	NO
Math_24	1327	0	00:15	No (0)	No (0)	NO	25	0	24:06	10/10 (10)	same (0)	same (0)	NO
Math_28	219	0	00:17	No (0)	No (0)	NO	0	0	00:30	0/30 (1)	same (0)	same (0)	NO
Math_33	1749	1	00:13	Yes (1)	No (0)	NO	19	0	10:30	28/30 (8)	same (0)	worse (-2.0)	NO
Math_40	831	71	00:16	Yes (71)	Yes (21)	NO	392	0	07:00	7/30 (8)	same (0)	same (0)	NO
Math_41	1224	0	00:06	No (0)	Yes (41)	NO	35	0	02:00	27/30 (27)	same (0)	improve (35.1)	NO
Math_42	1770	19	00:04	Yes (19)	No (0)	NO	2	0	03:54	24/30 (22)	same (0)	same (0)	NO
Math_50	1107	26	00:11	Yes (21)	Yes (45)	NO	23	1	04:36	28/30 (27)	improve (1.1)	improve (41.0)	NO
Math_57	651	0	00:03	No (0)	No (0)	NO	0	0	00:48	15/30 (4)	same (0)	same (0)	NO
Math_58	228	0	00:06	No (0)	No (0)	NO	7	0	00:20	2/30 (2)	same (0)	same (0)	NO
Math_69	897	0	00:01	No (0)	No (0)	NO	30	0	00:12	30/30 (21)	same (0)	same (0)	NO
Math_71	951	0	00:01	No (0)	Yes (56)	NO	17	0	00:24	25/30 (11)	same (0)	improve (53.0)	NO
Math_73	1035	0	00:01	No (0)	Yes (1)	NO	10	0	00:18	25/30 (24)	same (0)	improve (1)⋆	NO
Math_78	1014	0	00:01	No (0)	Yes (44)	NO	49	0	00:24	28/30 (16)	same (0)	improve (34.9)	NO
Math_80	1356	67	00:01	Yes (49)	No (0)	NO	29	1	00:54	29/30 (27)	worse (-17.9)	same (0)	NO
Math_81	1320	4	00:01	Yes (4)	Yes (35)	NO	30	0	00:24	23/30 (22)	same (0)	improve (35.0)⋆	NO
Math_82	510	0	00:01	No (0)	No (0)	NO	0	0	00:08	0/30 (1)	same (0)	same (0)	NO
Math_84	165	0	00:01	No (0)	No (0)	NO	0	0	00:06	0/30 (1)	same (0)	same (0)	NO
Math_85	798	0	00:01	No (0)	No (0)	NO	32	0	00:12	28/30 (11)	same (0)	same (0)	**YES**
Math_87	1866	14	00:01	Yes (13)	Yes (8)	NO	0	0	00:54	29/30 (29)	worse (-1)	improve (8.0)⋆	NO
Math_88	1890	11	00:01	Yes (11)	No (0)	NO	0	0	00:30	06/30 (7)	same (0)	same (0)	NO
Math_105	1353	7	00:09	Yes (7)	Yes (6)	NO	6	0	04:20	29/30 (30)	same (0)	improve (2.9)	NO
Time_4	2778	5	00:01	Yes (5)	Yes (6)	NO	0	0	00:54	23/30 (23)	improve (0.8)	improve (5.7)	NO
Time_7	1491	0	00:01	No (0)	Yes (11)	NO	12	0	00:54	12/30 (13)	same (0)	worse (-1)	NO
Time_11	1497	5	00:04	Yes (5)	No (0)	NO	7	0	01:36	0/30 (1)	same (0)	same (0)	NO
Time_14	687	0	00:01	No (0)	Yes (3)	NO	1	0	00:18	24/30 (23)	same (0)	improve (2.0)	NO
Time_16	1476	0	00:01	No (0)	Yes (6)	NO	5	0	00:24	1/30 (2)	same (0)	improve (1)	NO

The test generation results by running EvoSuite are shown in the two columns under the column *Tests*, among which the *#EvoTests* column shows the total number of tests generated by EvoSuite for all seeds and the *#Bug-expo* column shows the number of bug-exposing tests among all of the generated tests.

The results obtained by running just Nopol are shown in the columns under the column *Nopol*. The *Time* column shows the time used by Nopol to generate the initial patch. The *incomplete fix (#failing)* column shows what is the overfitting issue of incomplete fixing for the original Nopol patch. Each cell in this column is of the form X (Y), where X can be "Yes" or "No" and Y is a digit number. The "Yes" and "No" mean that the original Nopol patch has and does not have overfitting issue of incomplete fixing respectively. The digit number in parentheses shows the number of bug-exposing tests on which the original Nopol patch fails. Similarly, the *regression (#failing)* column tells what is the overfitting issue of regression introduction for the original Nopol patch, and each cell in this column is of the same form with the column *incomplete fix (#failing)*. The "Yes" and "No" for this column mean that the original Nopol patch has and does not have overfitting issue of regression introduction respectively. The digit number in parentheses shows the number of normal tests on which the original Nopol patch fails. Finally, the column *correctness* shows whether the original Nopol patch is correct, with "Yes" representing correct and "No" representing incorrect.

The results obtained by running Nopol+UnsatGuided are shown in the remaining columns under the column *Nopol+UnsatGuided*. The *#Removed* column shows the total number of removed generated tests during the run of Nopol+UnsatGuided for all seeds. The number of bug-exposing tests among the removed tests is shown in the column *#Removed Bug-expo*. The *Avg#Time* column shows the average time used by Nopol+UnsatGuided to generate the patch for each seed. The *Change ratio (#unique)* column is of the form X/Y (Z). Here Y is the number of different seeds used, X refers to the number of generated patches by Nopol+UnsatGuided that are different from the original Nopol patch, and Z is the number of distinct patches among all of the patches generated for all seeds.

The following two columns *fix completeness change (Avg#Removedinc)* and *regression change (Avg#Removedreg)* show the effectiveness of UnsatGuided in alleviating overfitting issue of incomplete fixing and regression introduction respectively. Each cell in these two columns is of the form X (Y), where X can be "improve", "worse", and "same" and Y is a digit number.

Compared with the original Nopol patch, the "improve", "worse", and "same" in column *fix completeness change (Avg#Removedinc)* mean that the new patch generated by running Nopol+UnsatGuided has less, more, and the same overfitting issue of incomplete fixing respectively. The digit number gives a more detailed information. In particular, it gives the average number of removed failing bug-exposing tests for the new patch generated by running Nopol+UnsatGuided compared with the original Nopol patch. In other words, the digital value is obtained by subtracting the average number of failing bug-exposing tests for the new patch generated by running Nopol+UnsatGuided from the number of failing bug-exposing tests for the original Nopol patch. A positive value is good, which shows that the new patch has less overfitting issue of incomplete fixing in a way. For example, a value of 1 says that the new patch does not exhibit overfitting issue of incomplete fixing anymore for a test case within I_{bug}.

Similarly, compared with the original Nopol patch, the "improve", "worse", and "same" in column *regression change (Avg#Removedreg)* mean that the new patch generated by running Nopol+UnsatGuided has less, more, and the same overfitting issue of regression introduction respectively. Compared with the original Nopol patch, the digit number in column *regression change (Avg#Removedreg)* gives the average number of removed failing normal tests for the new patch generated by running Nopol+UnsatGuided, and it equates to the value obtained by subtracting the average number of failing normal tests for the new patch generated by running Nopol+UnsatGuided from the number of failing normal tests for the original Nopol patch. Again, a positive value is good, which shows that the new patch has less overfitting issue of regression introduction in a way. For example, a value of 2 says that the new patch does not exhibit overfitting issue of regression introduction anymore for two test cases within $I_{correct}$.

Note for the patch generated using Nopol+UnsatGuided for a certain seed, the tests considered are all tests generated using all seeds for the corresponding program version. We average the results for all seeds of a certain program version and the resultant numbers are shown as digit numbers in the columns *fix completeness change (Avg#Removedinc)* and *regression change (Avg#Removedreg)*. Overall, *a positive digit number in these two columns shows an improvement: it means that overfitting issue of incomplete fixing or regression introduction has been alleviated after running UnsatGuided*. In addition, we use "perfect" to refer to the situation where for each seed of a certain program version, running *Nopol+UnsatGuided* with the seed will get a patch that will completely remove the overfitting issue of the original Nopol patch. The "perfect" results are illustrated with ($\star$).

Finally, the column *correctness* under the column *Nopol+UnsatGuided* shows whether the selected patch generated by running *Nopol+UnsatGuided* is correct, again with "Yes" representing correct and "No" representing incorrect.

4.5 **RQ1**: Prevalence of the Two Kinds of Overfitting Issues

We first want to measure the prevalence of overfitting issues of incomplete fixing and regression introduction among the patches generated by synthesis-based repair techniques.

We can see from the *incomplete fix (#failing)* and *regression (#failing)* columns under the column *Nopol* that for the 42 buggy versions that Nopol can generate an initial patch, overfitting can be observed for 26 buggy versions (when there exists "Yes" in either of these two columns).

Among the other 16 buggy versions for which we do not observe any kinds of overfitting issues, the manual analysis shows that the Nopol patches for two buggy versions (Lang_44 and Lang_55) are correct. However, the manual analysis shows that the Nopol patches for the remaining 14 buggy versions are incorrect, yet we do not observe any number of failing bug-exposing or normal tests for the programs patched with the patches generated by Nopol. This shows the limitation of automatic test case generation in covering the buggy input domain I_{bug} for real programs, which confirms a previous study (Shamshiri et al (2015)).

Among the 26 buggy versions for which we observe overfitting issues, the original Nopol patches for 13 buggy versions have the overfitting issue of incomplete fixing, the original Nopol patches for 19 buggy versions have the overfitting issue

of regression introduction, and the original Nopol patches for 6 buggy versions have both the overfitting issues of incomplete fixing and regression introduction. Thus, both the overfitting issues of incomplete fixing and regression introduction are common for the Nopol patches.

It can also be seen from Table 2 that the severity of overfitting differs from one patch to another as measured by the number of failing tests. Among the 13 patches that have overfitting issue of incomplete fixing, the number of failing bug-exposing tests is less than 3 for 3 patches (which implies the overfitting issue is relatively light), yet this number is larger than 20 for 3 patches (which implies the overfitting issue is relatively serious).

Similarly, for the 19 patches that have overfitting issue of regression introduction, the number of failing normal tests is less than 3 for 1 patch (which implies the overfitting issue is relatively light), yet this number is larger than 20 for 6 patches (which implies the overfitting issue is relatively serious).

Answer for RQ1: Both overfitting issues of incomplete fixing (13 patches) and regression introduction (19 patches) are common for the patches generated by Nopol.

4.6 **RQ2**: Effectiveness of UnsatGuided in Alleviating Overfitting Issues

We then want to assess the effectiveness of UnsatGuided. It can be seen from the column *Change ratio (#unique)* of Table 2 that for the 42 buggy versions that can be initially repaired by Nopol, the patches generated for 34 buggy versions have been changed at least for one seed after running Nopol+UnsatGuided. If we consider all executions (one per seed per buggy version), we obtain a total of 1220 patches with Nopol+UnsatGuided. Among the 1220 patches, 702 patches are different from the original patches generated by running Nopol only. Thus, UnsatGuided can significantly impact the output of the Nopol repair process. We will further investigate the quality difference between the new Nopol+UnsatGuided patches and the original Nopol patches.

The results for alleviating the two kinds of overfitting issues by running Nopol+ UnsatGuided are displayed in the columns *fix completeness change (Avg #Removedinc)* and *regression change (Avg#Removedreg)* of Table 2.

With regard to alleviating the overfitting issue of incomplete fixing, we can see from the column *fix completeness change (Avg#Removedinc)* that UnsatGuided has an effect on 4 buggy program versions (Math_50, Math_80, Math_87 and Time_4). For all those 4 buggy versions, the original Nopol patch already has the overfitting issue of incomplete fixing. With UnsatGuided, the overfitting issue of incomplete fixing has been alleviated in 2 cases (Math_50, Time_4) and worsened for 2 other cases (Math_80, Math_87). This means UnsatGuided is likely to have a minimal positive impact on alleviating overfitting issue of incomplete fixing and can possibly have a negative impact on it, confirming our analysis in Section 3. We will further discuss this point in RQ4 (Section 4.8).

In terms of alleviating overfitting issue of regression introduction, we can see from the column *regression change (Avg#Removedreg)* that UnsatGuided has an effect on 18 buggy program versions. Among the 18 original Nopol patches for these 18 buggy program versions, UnsatGuided has alleviated the overfitting issue

of regression introduction for 16 patches. In addition, for 6 buggy program versions, the overfitting issue of regression introduction of the original Nopol patch has been completely removed. These 6 cases are indicated with ($\star$) in Table 2. Meanwhile, UnsatGuided worsens the overfitting issue of regression introduction for two other original Nopol patches (Math_33 and Time_7). It can possibly happen as even though the repair constraint for input points within $I_{correct}$ has been somewhat strengthened (but not completely correct), yet the solution of the constraint happens to be more convoluted. Overall, with 16 positive versus 2 negative cases, UnsatGuided can be considered as effective in alleviating overfitting issue of regression introduction.

Answer for RQ2: UnsatGuided can effectively alleviate the overfitting issue of regression introduction (16/19 cases), but has minimal positive impact on reducing the overfitting issue of incomplete fixing. This results confirm our deductive analysis of the effectiveness of UnsatGuided in alleviating the two kinds of overfitting issues (Section 3).

4.7 **RQ3**: Impact of UnsatGuided on Patch Correctness

We will further assess the impact of UnsatGuided on the correctness of the patches. More specifically, we will assess 1) whether running Nopol+UnsatGuided destroys the already correct patches generated by Nopol (i.e., make them become incorrect) and 2) whether running Nopol+UnsatGuided can change an overfitting patch generated by Nopol into a completely correct one.

Can already correct patches be broken? The previous paper (Martinez et al (2016)) claims that running Nopol can generate correct patches for 5 buggy program versions Chart_5, Lang_44, Lang_55, Lang_58, and Math_50. However, for three of them (Chart_5, Lang_58, and Math_50), we can see from Table 2 that some EvoSuite tests fail on the original Nopol patches. Due to the use of different Nopol versions, the Nopol patch generated in this paper for Math_50 is different from that in (Martinez et al (2016)). We run the EvoSuite tests against the Nopol patch in (Martinez et al (2016)) and we also observe failing tests. To ensure the validity of the bug detection results, two authors of this paper have manually checked the correctness of the patches generated for these three buggy versions in the paper (Martinez et al (2016)). The overall results suggest that the original Nopol patches for these three program versions are not truly correct, which shows the inherent difficulty of manual analysis. For the other 2 buggy program versions (Lang_44 and Lang_55), there is no indication of overfitting and we consider the original Nopol patches as well as the new patches generated by running Nopol+UnsatGuided as correct. We now demonstrate why they can be considered as correct.

For Lang_44, the bug arises for a method which parses a string to a number (`String` to `int`, `long`, `float` or `double`) (see Figure 2). If the string (*val*) only contains the char *L* which specifies the type `long`, the method returns an *IndexOutOfBoundsException* (due to the expression `numeric.substring(1)` in the *if* condition) instead of the expected *NumberFormatException*, the other situations have already been correctly handled. The human patch adds a check at the beginning of the method to avoid this specific situation. The original Nopol patch sim-

```
// MANUAL PATCH
// if (val.length() == 1 && !Character.isDigit(val.charAt(0))) {
//  throw new NumberFormatException(val + " is not a valid number.");
// }
String numeric=val.substring(0, val.length()-1);
  ...
 switch (lastChar) {
  case 'L' :
    if (dec == null && exp == null && (numeric.charAt(0) == '-' &&
         isDigits(numeric.substring(1)) || isDigits(numeric))) {
       try {
         return createLong(numeric);
        } catch (NumberFormatException nfe) { }
       return createBigInteger(numeric);
     }
    throw new NumberFormatException(val + "is not a valid number.");
   case 'f' : ...
```

Fig. 2 Code snippet of buggy program version Lang_44.

plifies the *if* condition to (`dec == null && exp == null`) and relies on checks available in the called method (`createLong(String val)`), which will return a *NumberFormatException* if the format of input *val* is illegal. Note the deleted predicate `(numeric.charAt(0)=='-' && isDigits(numeric.substring(1)) || isDigits(numeric))` is used to check whether the variable *numeric* is a legal format of number, and a *NumberFormatException* will be thrown if not. Consequently, for the specific input L and other inputs which are not legal forms of number, the desired *NumberFormatException* will also be thrown after the condition is simplified. Among the 30 seeds, running Nopol+UnsatGuided with 27 seeds will get the same patch as the original Nopol run. For the other 3 seeds, running Nopol+UnsatGuided will all get the patch which adds the precondition `if(1 < val.length())` before the *if* condition. After adding this precondition, the *if* condition is executed only when the length of the string is larger than 1. If this precondition is not respected, the program throws the expected exception. Thus, both the original Nopol patch and the new patch generated by running Nopol+UnsatGuided are semantically equivalent to the human patch.

For Lang_55, the bug arises for a utility class for timing (see Figure 3). As discussed in Martinez et al (2016), the bug appears when the user stops a suspended timer and if so, the stop time saved by the suspend action is overwritten by the stop action. To fix the bug, the assignment of the variable *stopTime* should be executed only when the state of the timer is running. The human patch adds a precondition which checks whether the state of the timer is running. The original Nopol patch and the patch generated by running Nopol+UnsatGuided (running the 30 seeds all get the same patch) both also add preconditions. Note the method `stop()` should be executed only when the state of the timer is suspended or running (see the if condition inside the method `stop()`), otherwise an exception will be thrown. Thus, the precondition `if (this.runningState!= STATE_SUSPENDED)` obtained by running Nopol means the state of the timer is running. Meanwhile, given the two possible states–suspended or running, the precondition `if (this.stopTime <= this.startTime)` obtained by running Nopol+UnsatGuided can only be true

```
public void stop() {
      if(this.runningState != STATE_RUNNING && this.runningState !=
          STATE_SUSPENDED) {
         throw new IllegalStateException("Stopwatch is not running. ");
      }
// MANUAL PATCH:
// if (this.runningState == STATE_RUNNING)
// NOPOL PATCH:
// if (this.runningState!= STATE_SUSPENDED)
// NOPOL+UnsatGuided PATCH:
// if(this.stopTime <= this.startTime)
         stopTime = System.currentTimeMillis();
      this.runningState = STATE_STOPPED;
}
```

Fig. 3 Code snippet of buggy program version Lang_55.

when the state of the timer is running according to the logic of the utility class. Consequently, both of the two added preconditions are semantically equivalent to the precondition added by human beings.

In summary, the correct patches generated by Nopol are still correct for all seeds after running Nopol+UnsatGuided.

Can an overfitting patch be changed into a correct one?

It has already been shown that running Nopol+UnsatGuided can significantly change the original Nopol patch and can effectively alleviate the overfitting issue of regression introduction in the original Nopol patch. We want to further explore whether an overfitting patch can be changed into a correct one after running Nopol+UnsatGuided. Comparing the two *correctness* columns under the column *Nopol* and column *Nopol+UnsatGuided*, we can see that there exists one buggy version (Math_85) for which the original Nopol patch is incorrect but the sampled patch generated by running Nopol+UnsatGuided is correct.

For Math_85, the bug arises as the value of a condition is not handled appropriately (see Figure 4). The human patch changes the binary relational operator from "$>=$" to "$>$", i.e., replacing `if (fa * fb >= 0.0)` with `if (fa * fb > 0.0)`. The original Nopol patch adds a precondition `if (fa * fb < 0.0)` before the *if* condition in the code, which in turn will result in a self-contradictory condition and is thus incorrect. The sampled Nopol+UnsatGuided patch is adding a precondition `if (fa * fb != 0.0)` before the *if* condition, which equates to the human patch semantically and is thus correct. After further checking the results for this buggy version across all 30 seeds, we find that the generated Nopol+UnsatGuided patch is the same as this patch for 21 seeds. This example shows that UnsatGuided can possibly change an original overfitting Nopol patch into a correct one.

Answer for RQ3: UnsatGuided does not break any already correct Nopol patch. Furthermore, UnsatGuided can change an overfitting Nopol patch into a correct one. This is in line with our analysis of the impact of UnsatGuided on patch correctness.

```
if (fa * fb >= 0.0 ) {
      throw new ConvergenceException(
         ...
      );
  }
```

Fig. 4 Code snippet of buggy program version Math_85.

4.8 **RQ4**: Handling of Bug-exposing Tests

As we have seen in Section 3.4, the major challenge of using automatic test generation in the context of repair is the handling of bug-exposing tests. However, bug-exposing tests are not always generated. Now we concentrate on the 17 buggy program versions which contain at least one bug-exposing test, i.e., rows in Table 2 with the value of *#Bug-expo* larger than 0.

For 4 bugs (Chart_5, Lang_44, Lang_51, Lang_63), UnsatGuided works perfectly because it removes all bug-exposing tests. Let us now explain what happens in those cases. The column *incomplete fix (#failing)* shows that for these 4 buggy versions, the original Nopol patch does not fail on any of the bug-exposing tests, which implies that the initial repair constraint established using the manually written test suite is strong and is likely to have reflected the desired behaviors for input points within I_{bug} well. In this case, the additional repair constraints enforced by the bug-exposing tests have contradictions with the initial repair constraint and UnsatGuided indeed removes them, as it is designed for. If we do not take care of this situation and directly use all of the automatically generated tests without any removal technique, we are likely to lose the correct repair constraint and the acceptable patch with no overfitting issue of incomplete fixing.

For the other 13 buggy program versions, the bug-exposing tests are either not removed at all (11 cases) or partially removed (2 cases, Math_50 and Math_80). The column *incomplete fix (#failing)* shows that for these 13 buggy versions, the original Nopol patch already fails on some of the bug-exposing tests, which implies that the initial repair constraint established using the manually written test suite does not fully reflect the desired behaviors for input points within I_{bug}. Consequently, no contradiction happens during the synthesis process and these bug-exposing tests are not recognized and kept. Now, recall that we have explained in Section 3.4 that the presence of remaining bug-exposing tests does not necessarily mean worsened overfitting issue of incomplete fixing. Interestingly, this can be shown in our evaluation: for 9 bugs, the overfitting issue of incomplete fixing remains the same; for 2 bugs (Math_50 and Time_4), the overfitting issue of incomplete fixing is reduced (the digit value in column *fix completeness change (Avg#Removedinc)* is larger than 0); and for 2 other bugs (Math_80 and Math_87), the overfitting issue of incomplete fixing is worsened (the digit value in column *fix completeness change (Avg#Removedinc)* is smaller than 0). To sum up, the unremoved bug-exposing tests do not worsen overfitting issue of incomplete fixing for the original Nopol patch in the majority of cases (11/13 cases).

Finally, let us check whether the presence of kept bug-exposing tests will have an impact on the capability of UnsatGuided in alleviating overfitting issue of regression introduction. For the 13 buggy program versions with at least one remain-

ing bug-exposing test, we see that UnsatGuided is still able to alleviate overfitting issue of regressions introduction. This is the case for 5 bug versions: Math_50, Math_81, Math_87, Math_105, and Time_4. This result confirms our qualitative analysis, i.e., the unremoved bug-exposing tests will not impact the effectiveness of UnsatGuided in alleviating overfitting issue of regression introduction.

Answer for RQ4: When bug-exposing tests are generated, UnsatGuided does not suffer from a drop in effectiveness: the overfitting issue of incomplete fixing is not worsened in the majority of cases, and the capability of alleviating overfitting issue of regression introduction is kept.

4.9 **RQ5**: Time Overhead

The time cost of an automatic program repair technique should be manageable for being used in industry. We now discuss the time overhead incurred by UnsatGuided.

To see the time overhead incurred, we compare the *Time* column under the column *Nopol* with the *Avg#Time* column under the column *Nopol+UnsatGuided*. First, we see that the approach UnsatGuided incurs some time overhead. Compared with the original repair time used by Nopol to find a patch, the average time used by running Nopol+UnsatGuided to get the patch is much longer. Second, the time overhead incurred is acceptable in many cases. Among the 42 buggy versions that can initially be repaired by Nopol, the average repair time used by running Nopol+UnsatGuided to get the patch is less than or equal to 1 hour for 28 buggy versions, which is arguably acceptable. Finally, we observe that the time overhead incurred can be extremely large sometimes. For 3 buggy versions (Chart_26, Math_24, and Math_33), running Nopol+UnsatGuided will cost more than 10 hours to get the patch on average. In particular, the average time used by running Nopol+UnsatGuided to get the patch for Math_24 is 24.1 hours. The synthesis process of Nopol is slow for those cases and the synthesis process is invoked for each generated test as required by UnsatGuided, thus the large amount of time cost is imaginable. To reduce the time overhead, future work will explore advanced patch analysis to quickly discard useless tests and identify generated tests that have the potential to improve the patch.

Answer for RQ5: UnsatGuided incurs a time overhead even though the overhead is arguably acceptable in many cases. To reduce the time overhead, more advanced techniques can be employed to analyze the automatically generated tests and discard useless ones.

4.10 Threats to Validity

We use 224 faults of 4 java programs from Defects4J in this study and one threat to external validity is whether our results will hold for other benchmarks. However, Defects4J is the most recent and comprehensive dataset of java bugs currently available, and is developed with the aim of providing real bugs to enable reproducible studies in software testing research. Besides, Defects4J has been extensively used as the evaluation subjects by recent research work in software testing

(B. Le et al (2016); Pearson et al (2017); Laghari et al (2016)), and in particular by work in automated program repair (Martinez et al (2016); Xiong et al (2017)). Another threat to external validity is that we evaluate the approach UnsatGuided by viewing Nopol as the representative for synthesis-based repair techniques, and doubts may arise whether the results will generalize to other synthesis-based repair techniques. Nopol, however, is the only open-source synthesis-based repair technique that targets modern java code and can effectively repair real-life faults in real-world programs. A final threat to external validity is that only one automatic test case generation tool, i.e., EvoSuite, is used in the study. But EvoSuite is the state-of-art open source java unit test case generation tool and can target a specific java class as required by the proposed approach. Moreover, we run EvoSuite 30 times with different random seeds to account for the randomness of EvoSuite. Overall, the evaluation results are in line with our analysis of the effectiveness of UnsatGuided in alleviating different kinds of overfitting issues, and we believe the results can be generalized.

A potential threat to internal validity is that we manually check the generated patches to investigate the impact of UnsatGuided on patch correctness. We used the human patch as the correctness baseline and the human patch is also used to help us understand the root cause of the bug. This process may introduce errors. To reduce this threat as much as possible, the results reported in this paper are checked and confirmed by two authors of the paper. In addition, the whole artifact related to this paper is made available online to let readers gain a more deep understanding of our study and analysis.

5 Conclusion

Much progress has been made in the area of test suite based program repair over the recent years. However, test suite based repair techniques suffer from the overfitting problem. In this paper, we deeply analyze the overfitting problem in program repair and identify two kinds of overfitting issues: incomplete fixing and regression introduction. We further define three kinds of overfitting patches based on the overfitting issues that a patch has. These characterizations of overfitting will help the community to better understand and design techniques to defeat the overfitting problem in program repair. We also propose an approach called UnsatGuided, which aims to alleviate the overfitting problem for synthesis-based repair techniques. The approach uses additional automatically generated tests to strengthen the repair constraint used by synthesis-based repair techniques. We analyze the effectiveness of UnsatGuided with respect to alleviating different kinds of overfitting issues. The general usefulness of automatic test case generation in alleviating overfitting problem is also discussed. An evaluation on the 224 bugs of the Defects4J repository has confirmed our analysis and shows that UnsatGuided is effective in alleviating overfitting issue of regression introduction.

References

Almasi MM, Hemmati H, Fraser G, Arcuri A, Benefelds J (2017) An industrial evaluation of unit test generation: Finding real faults in a financial application. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, IEEE Press, Piscataway, NJ, USA, ICSE-SEIP '17, pp 263–272, DOI 10.1109/ICSE-SEIP.2017.27, URL https://doi.org/10.1109/ICSE-SEIP.2017.27

Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '11, pp 1–10, DOI 10.1145/1985793.1985795, URL http://doi.acm.org/10.1145/1985793.1985795

B Le TD, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp 177–188, DOI 10.1145/2931037.2931049, URL http://doi.acm.org/10.1145/2931037.2931049

Baresi L, Lanzi PL, Miraz M (2010) Testful: an evolutionary test approach for java. In: Software testing, verification and validation (ICST), 2010 third international conference on, IEEE, pp 185–194

Brumley D, cker Chiueh T, Johnson R, Lin H, Song D (2007) Rich: Automatically protecting against integer-based vulnerabilities. In: In Symp. on Network and Distributed Systems Security

Cadar C, Dunbar D, Engler DR, et al (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224

Csallner C, Smaragdakis Y (2004) Jcrasher: an automatic robustness tester for java. Software: Practice and Experience 34(11):1025–1050

Durieux T, Cornu B, Seinturier L, Monperrus M (2017) Dynamic patch generation for null pointer exceptions using metaprogramming. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 349–358, DOI 10.1109/SANER.2017.7884635

Evans RB, Savoia A (2007) Differential testing: A new approach to change detection. In: The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, ACM, New York, NY, USA, ESEC-FSE companion '07, pp 549–552, DOI 10.1145/1295014.1295038, URL http://doi.acm.org/10.1145/1295014.1295038

Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE '11, pp 416–419, DOI 10.1145/2025113.2025179, URL http://doi.acm.org/10.1145/2025113.2025179

Gao Q, Xiong Y, Mi Y, Zhang L, Yang W, Zhou Z, Xie B, Mei H (2015) Safe memory-leak fixing for c programs. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 1, pp 459–470, DOI 10.1109/ICSE.2015.64

Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: ACM Sigplan Notices, ACM, vol 40, pp 213–223

Goues CL, Nguyen T, Forrest S, Weimer W (2012) Genprog: A generic method for automatic software repair. IEEE Trans Software Eng 38(1):54–72

Gu Z, Barr ET, Hamilton DJ, Su Z (2010) Has the bug really been fixed? In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA, ICSE '10, pp 55–64, DOI 10.1145/1806799.1806812, URL http://doi.acm.org/10.1145/1806799.1806812

Islam M, Csallner C (2010) Dsc+mock: A test case + mock class generator in support of coding against interfaces. In: Proceedings of the Eighth International Workshop on Dynamic Analysis, ACM, New York, NY, USA, WODA '10, pp 26–31, DOI 10.1145/1868321.1868326, URL http://doi.acm.org/10.1145/1868321.1868326

Jha S, Gulwani S, Seshia SA, Tiwari A (2010) Oracle-guided component-based program synthesis. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA, ICSE '10, pp 215–224, DOI 10.1145/1806799.1806833, URL http://doi.acm.org/10.1145/1806799.1806833

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated

Software Engineering, ACM, New York, NY, USA, ASE '05, pp 273–282, DOI 10.1145/1101908.1101949, URL http://doi.acm.org/10.1145/1101908.1101949

Just R, Jalali D, Ernst MD (2014a) Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, pp 437–440

Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014b) Are mutants a valid substitute for real faults in software testing? In: FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering, Hong Kong, pp 654–665

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, pp 802–811

Laghari G, Murgia A, Demeyer S (2016) Fine-tuning spectrum based fault localisation with frequent method item sets. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE 2016, pp 274–285, DOI 10.1145/2970276.2970308, URL http://doi.acm.org/10.1145/2970276.2970308

Le XBD, Chu DH, Lo D, Le Goues C, Visser W (2017a) S3: Syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2017, pp 593–604, DOI 10.1145/3106237.3106309, URL http://doi.acm.org/10.1145/3106237.3106309

Le XBD, Thung F, Lo D, Le Goues C (2017b) Overfitting in semantics-based automated program repair

Liu C, Fei L, Yan X, Han J, Midkiff SP (2006) Statistical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering 32(10):831–848, DOI 10.1109/TSE.2006.105

Liu X, Zeng M, Xiong Y, Zhang L, Huang G (2017) Identifying patch correctness in test-based automatic program repair. arXiv preprint arXiv:170609120

Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2015, pp 166–178, DOI 10.1145/2786805.2786811, URL http://doi.acm.org/10.1145/2786805.2786811

Long F, Rinard M (2016) Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '16, pp 298–312, DOI 10.1145/2837614.2837617, URL http://doi.acm.org/10.1145/2837614.2837617

Long F, Amidon P, Rinard M (2017) Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 727–739

Martinez M, Monperrus M (2016) Astor: A program repair library for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2016, pp 441–444, DOI 10.1145/2931037.2948705, URL http://doi.acm.org/10.1145/2931037.2948705

Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2016) Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empirical Software Engineering pp 1–29

Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, pp 448–458

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '16, pp 691–701, DOI 10.1145/2884781.2884807, URL http://doi.acm.org/10.1145/2884781.2884807

Monperrus M (2017) Automatic Software Repair: a Bibliography. ACM Computing Surveys URL https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf

Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '13, pp 772–781, URL http://dl.acm.org/

citation.cfm?id=2486788.2486890
Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, ACM, pp 815–816
Park S, Hossain B, Hussain I, Csallner C, Grechanik M, Taneja K, Fu C, Xie Q (2012) Carfast: Achieving higher statement coverage faster. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, p 35
Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '17, pp 609–620, DOI 10.1109/ICSE.2017.62, URL https://doi.org/10.1109/ICSE.2017.62
Pei Y, Furia CA, Nordio M, Wei Y, Meyer B, Zeller A (2014) Automated fixing of programs with contracts. IEEE Transactions on Software Engineering 40(5):427–449, DOI 10.1109/TSE.2014.2312918
Perkins JH, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G, Wong WF, Zibin Y, Ernst MD, Rinard M (2009) Automatically patching errors in deployed software pp 87–102, DOI 10.1145/1629575.1629585, URL http://doi.acm.org/10.1145/1629575.1629585
Prasetya ISWB (2014) T3, a Combinator-Based Random Testing Tool for Java: Benchmarking, Springer International Publishing, Cham, pp 101–110. DOI 10.1007/978-3-319-07785-7_7, URL http://dx.doi.org/10.1007/978-3-319-07785-7_7
Păsăreanu CS, Rungta N (2010) Symbolic pathfinder: Symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE '10, pp 179–180, DOI 10.1145/1858996.1859035, URL http://doi.acm.org/10.1145/1858996.1859035
Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE 2014, pp 254–265, DOI 10.1145/2568225.2568254, URL http://doi.acm.org/10.1145/2568225.2568254
Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of ISSTA, ACM
Sen K, Marinov D, Agha G (2005) Cute: a concolic unit testing engine for c. In: ACM SIGSOFT Software Engineering Notes, ACM, vol 30, pp 263–272
Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 201–211, DOI 10.1109/ASE.2015.86
Shaw A, Doggett D, Hafiz M (2014) Automatically fixing c buffer overflows using program transformations. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp 124–135, DOI 10.1109/DSN.2014.25
Smith EK, Barr ET, Le Goues C, Brun Y (2015) Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, pp 532–543
Taneja K, Xie T (2008) Diffgen: Automated regression unit-test generation. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering pp 407–410
Tian Y, Ray B (2017) Automatically diagnosing and repairing error handling bugs in c. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA, ESEC/FSE 2017, pp 752–762, DOI 10.1145/3106237.3106300, URL http://doi.acm.org/10.1145/3106237.3106300
Tillmann N, De Halleux J (2008) Pex: White box test generation for .net. In: Proceedings of the 2Nd International Conference on Tests and Proofs, Springer-Verlag, Berlin, Heidelberg, TAP'08, pp 134–153, URL http://dl.acm.org/citation.cfm?id=1792786.1792798
Tonella P (2004) Evolutionary testing of classes. SIGSOFT Softw Eng Notes 29(4):119–128, DOI 10.1145/1013886.1007528, URL http://doi.acm.org/10.1145/1013886.1007528
Wei Y, Pei Y, Furia CA, Silva LS, Buchholz S, Meyer B, Zeller A (2010) Automated fixing of programs with contracts. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA '10, pp 61–72, DOI 10.1145/1831708.1831716, URL http://doi.acm.org/10.1145/1831708.1831716

Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: Models and first results. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 356–366, DOI 10.1109/ASE.2013.6693094

Xie T (2006) Augmenting automatically generated unit-test suites with regression oracle checking. In: Proceedings of the 20th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg, ECOOP'06, pp 380–403, DOI 10.1007/11785477_23, URL http://dx.doi.org/10.1007/11785477_23

Xin Q, Reiss SP (2017) Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2017, pp 226–236, DOI 10.1145/3092703.3092718, URL http://doi.acm.org/10.1145/3092703.3092718

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE '17, pp 416–426, DOI 10.1109/ICSE.2017.45, URL https://doi.org/10.1109/ICSE.2017.45

Xuan J, Martinez M, Demarco F, Clément M, Lamelas S, Durieux T, Le Berre D, Monperrus M (2016) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Transactions on Software Engineering DOI 10.1109/TSE.2016.2560811, URL https://hal.archives-ouvertes.fr/hal-01285008/document

Yang J, Zhikhartsev A, Liu Y, Tan L (2017) Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, pp 831–841

Yi J, Tan SH, Mechtaev S, Böhme M, Roychoudhury A (2017) A correlation study between automated program repair and test-suite metrics. Empirical Software Engineering pp 1–32

Yu Z, Hu H, Bai C, Cai KY, Wong WE (2011) Gui software fault localization using n-gram analysis. In: 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering, pp 325–332, DOI 10.1109/HASE.2011.29

Yu Z, Bai C, Cai KY (2013) Mutation-oriented test data augmentation for gui software fault localization. Inf Softw Technol 55(12):2076–2098, DOI 10.1016/j.infsof.2013.07.004, URL http://dx.doi.org/10.1016/j.infsof.2013.07.004

Yu Z, Bai C, Cai KY (2015) Does the failing test execute a single or multiple faults?: An approach to classifying failing tests. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, IEEE Press, Piscataway, NJ, USA, ICSE '15, pp 924–935, URL http://dl.acm.org/citation.cfm?id=2818754.2818866

Yu Z, Martinez M, Danglot B, Durieux T, Monperrus M (2017) Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness. Tech. Rep. 1703.00198v1, Arxiv, URL https://arxiv.org/pdf/1703.00198

Zhang X, Gupta N, Gupta R (2006) Locating faults through automated predicate switching. In: Proceedings of the 28th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '06, pp 272–281, DOI 10.1145/1134285.1134324, URL http://doi.acm.org/10.1145/1134285.1134324